Industrial automation systems and integration — Product data representation and exchange — Part 14: Description methods: The EXPRESS-X Language Reference Manual

1. Scope

This part of ISO 10303 defines a language by which relationships between data defined by models in the EXPRESS language can be specified. The language is called EXPRESS-X.

EXPRESS-X is a structural data mapping language. It consists of language elements that allow an unambiguous specification of the relationship between models.

The following are within the scope of this part of ISO 10303:

- Mapping data defined by one EXPRESS model to data defined by another EXPRESS model.
- Mapping data defined by one version of an EXPRESS model to data defined by another version of an EXPRESS model, where the two schemas have different names.
- Specification of requirements for data translators for data sharing and data exchange applications.
- Specification of alternate views of data defined by an EXPRESS model.
- An alternate notation for application protocol mapping tables.
- Bi-directional mappings where mathematically possible.
- Specification of constraints evaluated against data produced by mapping.

The following are outside the scope of this part of ISO 10303:

- Mapping of data defined using means other than EXPRESS.
- Identification of the version of an EXPRESS schema.
- Graphical representation of constructs in the EXPRESS-X language.

2. Normative references

The following standards contain provisions that, through reference in this text, constitute provisions of this part of ISO 10303. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 10303-1:1994, *Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles.*

ISO standard 10303 part(11) version (3), *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual*.

3. Definitions

3.1 Terms defined in ISO 10303-1

This part of ISO 10303 makes use of the following terms defined in ISO 10303-1.

- data;
- information;
- information model.

3.2 Terms defined in ISO 10303-11

This part of ISO 10303 makes use of the following terms defined in ISO 10303-11.

- complex entity data type;
- complex entity (data type) instance;
- constant;
- entity;
- entity data type;
- entity (data type) instance;

- instance;
- partial complex entity data type;
- partial complex entity value;
- population;
- simple entity (data type) instance;
- subtype/supertype graph;
- token;
- value.

3.3 Other definitions

3.3.1 binding extent: a set of binding instances constructed from instances in the source data sets and view extents as required by the FROM language element of the VIEW or MAP declaration.

3.3.2 binding instance: an element of a binding extent.

3.3.3 source data set: a collection of entity instances where each entity instance conforms to an entity data type defined in the associated schema, and the collection conforms to the constraints of the schema.

3.3.4 target data set: a collection of entity instances produced by means of mapping.

3.3.5 map: the declaration of a relationship between data of one or more source entity types or view data types and data of one or more target entity types.

3.3.6 network mapping: a mapping to many target entity instances.

3.3.7 qualified binding extent: a subset of the binding extent consisting of only those binding instances satisfying the selection criteria of the view/map declaration.

3.3.8 selection criteria: EXPRESS logical expressions used to identify the qualified binding extent from a binding extent.

3.3.9 source extent: a view extent or entity population used to create binding extent.

3.3.10 view: an alternative organization of the information in an EXPRESS model.

3.3.11 view data type: the representation of a view.

3.3.12 view data type instance: a named unit of information that is a member of the view extent established by a view data type.

3.3.13 view extent: an aggregate of view data type instances that contains all instances that can be constructed from the qualified binding extent.

4. Conformance requirements

4.1 Formal specifications written in EXPRESS-X

4.1.1 Lexical language

A formal specification written in EXPRESS-X shall be consistent with a given level as specified below. A formal specification is consistent with a given level when all checks identified for that level as well as all lower levels are verified for the specification.

Levels of checking

Level 1: Reference checking. This level consists of checking the formal specification to ensure that it is syntactically and referentially valid. A formal specification is syntactically valid if it matches the syntax generated by expanding the primary syntax rule (`syntax`) given inAnnex A. A formal specification is referentially valid if all references to EXPRESS-X items are consistent with the scope and visibility rules defined in clauses 10 and11.

Level 2: Type checking. This level consists of Level 1 checking and checking the formal specification to ensure that it is consistent with the following:

- expressions shall comply with the rules specified in clause12 and in ISO 10303-11:1994 clause12;
- assignments shall comply with the rules specified in ISO 10303-11:1994 clause 13.3.

Level 3: Value checking. This level consists of Level 2 checking and checking the formal specification to ensure that it is consistent with statements of the form, ‘A shall be greater than B’, as specified in clause7 to14 of ISO 10303-11:1994. This is limited to those places where both A and B can be evaluated from literals and/or constants.

Level 4: Complete checking. This level consists of checking the formal specification to ensure that it is consistent with all stated requirements as specified in this part of ISO 10303 and of ISO 10303-11:1994.

4.2 Implementations of EXPRESS-X

4.2.1 EXPRESS-X language parser

An implementation of an EXPRESS-X language parser shall be able to parse any formal specification written in EXPRESS-X consistent with the conformance class associated with that implementation. An EXPRESS-X language parser shall be said to conform to a particular checking level (as defined in4.1.1) if it can apply all checks required by that level (and any level below it) to a formal specification written in EXPRESS-X.

The implementor of an EXPRESS-X language parser shall state all constraints that the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented for the purpose of conformance testing.

4.2.2 EXPRESS-X mapping engine

An implementation of an EXPRESS-X mapping engine shall be able to evaluate and/or execute any formal specification written in EXPRESS-X, consistent with the conformance class associated with that implementation. The execution and/or evaluation of a mapping is relative to one or more source data sets; the specification of how these data sets are made available to the mapping engine is outside the scope of this part of ISO10303.

The implementor of an EXPRESS-X mapping engine shall state any constraints that the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented for the purpose of conformance testing.

4.3 Conformance classes

An implementation shall be said to conform to conformance class1 if it processes all the declarations that may appear in a SCHEMA_VIEW declaration.

An implementation shall be said to conform to conformance class2 if it processes all the declarations that may appear in this part of ISO 10303.

5. Language specification syntax

The notation used to present the syntax of the EXPRESS-X language is defined in this clause.

The full syntax for the EXPRESS-X language is given in AnnexA. Portions of those syntax rules are reproduced in various clauses to illustrate the syntax of a particular statement. Those portions are not always complete. It will sometimes be necessary to consult AnnexA for the missing rules. The syntax portions within this part of ISO 10303 are presented in a box. Each rule within the syntax box has a unique number toward the left margin for use in cross-references to other syntax rules.

The syntax of EXPRESS-X is defined in a derivative of Wirth Syntax Notation (WSN).

NOTE — See annex B for a reference describing Wirth Syntax Notation.

The notational conventions and WSN defined in itself are given below.

```
syntax= { production } .
production= identifier '=' expression '.' .
expression= term { '|' term } .
term= factor { factor } .
factor= identifier | literal | group | option | repetition .
identifier= character { character } .
literal= '''' character { character } '''' .
group= '(' expression ')' .
option= '[' expression ']' .
repetition= '{' expression '}' .
```

– The equal sign `'='` indicates a production. The element on the left is defined to be the combination of the elements on the right. Any spaces appearing between the elements of a production are meaningless unless they appear within a literal. A production is terminated by a period `'.'`.

– The use of an identifier within a factor denotes a nonterminal symbol that appears on the left side of another production. An identifier is composed of letters, digits, and the underscore character. The keywords of the language are represented by productions whose identifier is given in uppercase characters only.

– The word literal is used to denote a terminal symbol that cannot be expanded further. A literal is a sequence of characters enclosed in apostrophes. For an apostrophe to appear in a literal it must be written twice, i.e., `''''`.

– The semantics of the enclosing braces are defined below:

  • curly brackets `'{ }'` indicates zero or more repetitions;

  • square brackets `'[ ]'` indicates optional parameters;

  • parenthesis `'( )'` indicates that the group of productions enclosed by parenthesis shall be used as a single production;

  • vertical bar `'|'` indicates that exactly one of the terms in the expression shall be chosen.

The following notation is used to represent entire character sets and certain special characters which are difficult to display:

- **\a** represents any character from ISO/IEC10646-1;
- **\n** represents a newline (system dependent) (see clause 7.1.5.2 of ISO 10303-11:1994).

6. Basic language elements

6.1 Overview

This clause specifies the basic elements from which an EXPRESS-X mapping specification is composed: the character set, remarks, symbols, reserved words, identifiers, and literals.

The basic language elements of EXPRESS-X are those of the EXPRESS language defined in Clause 7 of ISO10303-11, with the exceptions noted below.

6.2 Reserved words

The reserved words of EXPRESS-X are the keywords and the names of built-in constants, functions, and procedures. Any reserved word in EXPRESS (ISO10303-11:1994) shall also be a reserved word in EXPRESS-X. The reserved words shall not be used as identifiers. The additional reserved words of EXPRESS-X are described below.

In the case that a legal EXPRESS identifier is a reserved word in EXPRESS-X, schemas using that identifier can be mapped by renaming the conflicting identifier using the AS keyword in the REFERENCE language element.

In addition to the keywords of EXPRESS defined in ISO10303-11:1994, the following are keywords of EXPRESS-X.

Table 1 — Additional EXPRESS-X keywords

END_SCHEMA_MAP	EACH	END_TYPE_MAP	END_MAP
MAP	END_SCHEMA_VIEW	IMPORT_MAPPING	END_VIEW
SOURCE	IDENTIFIED_BY	SCHEMA_MAP	SCHEMA_VIEW
	PARTITION	TYPE_MAP	VIEW
	TARGET		

7. Data types

7.1 Overview

The data types defined here as well as those defined in the EXPRESS language (clause8 of ISO 10303-11:1994) are provided as part of the language.

Every view attribute has an associated data type.

7.2 View data type

View data types are established by view declarations (see clause 9.3). A view data type is assigned an identifier in the defining schema map or schema view. The view data type is referenced by this identifier.

Syntax:

```
127 view_reference = [ ( schema_map_ref | schema_view_ref ) '.' ] view_ref
    .
```

Rules and restrictions:

a) view_ref shall be a reference to a view visible in the current scope.

b) view_ref shall not refer to a return view (clause9.3.5).

EXAMPLE 1 — following declaration defines a view data type named circle.

```
VIEW circle;
  FROM (e : ellipse);
  WHERE (e.major_axis = e.minor_axis);
  SELECT
    radius : REAL := e.minor_axis;
    center : point := e.center;
END_VIEW;
```

8. Fundamental principles

8.1 Overview

The reader of this document is assumed to be familiar with the following concepts, in addition to the concepts described in clause5 of ISO10303-11:1994.

EXPRESS-X provides for the specification of:

- alternative views of the data described by an information model described in EXPRESS;
- the transformation of data described by elements of source EXPRESS models into data described by elements of target EXPRESS models.

A `SCHEMA_MAP` provides declarations for the specification of the former and latter.

A `SCHEMA_VIEW` provides declarations for the specification of the former.

NOTE — A SCHEMA_VIEW may be transformed into an EXPRESS model as described in Annex B.

The specification of a type map defines how data described by EXPRESS defined types may be transformed between the source and target models.

EXPRESS function and procedure specifications may form part of an EXPRESS-X schema in order to support the definition of views, maps, or type maps.

8.2 Typographical conventions

In this specification a binding instance is denoted as an ordered set of entity / view instance name separated by commas "," and enclosed in *angle brackets*, "<>". Entity instance names are defined in ISO standard 10303 part(21) clause7.3.4. View instance names are specified using the same syntax.

EXAMPLE 2 — Given the view declaration:

```
VIEW example;
  FROM p: person, o : organization;
...
END_VIEW;
```

the following may be binding instances:

```
<#1,#31>
<#2,#32>.
```

These binding instances may correspond to the following data presented as entity instances as defined in ISO standard 10303 part (21):

```
#1=person('James','Smith');
#2=person('Fredrick','Jones');
#31=organization('Engineering');
#32=organization('Sales');
```

In this specification the data referenced by a binding extent may be presented in tabular form where the left-most column identifies the binding instance. The uppermost column headings, excluding the left-most column, identify express entity types or view data types. The lower headings identify the names of attributes corresponding to the entity identified in the uppermost column under which it falls, or when the heading cell contains '#', the entity instance name.

EXAMPLE 3 — This example illustrates the use of tables to depict a binding extent. The concept of a binding extent is defined in subsequent clauses and is not necessary to understand the example. The example uses the data defined in example2 and the following EXPRESS schema:

```
SCHEMA example_3;
ENTITY person;
  first_name : STRING;
  last_name : STRING;
END_ENTITY;
ENTITY organization;
  department_name : STRING;
END_ENTITY;
END_SCHEMA;
```

Binding Instance	person			organization	
	#	first_name	last_name	#	department_name
<#1,#31>	#1	'James'	'Smith'	#31	'Engineering'
<#1,#32>	#1	'James'	'Smith'	#32	'Sales'
<#2,#31>	#2	'Fredrick'	'Jones'	#31	'Engineering'
<#2,#32>	#2	'Fredrick'	'Jones'	#32	'Sales'

In this specification a view instance, target entity or target entity network corresponding to particular binding instance is denoted by the name of the view declaration (`view_id`) or map declaration (`map_id`) of which it is a member followed by a *left parenthesis* '(', followed by the binding instance, followed by a *right parenthesis* ')'.

8.3 Binding process

This specification defines a language and an execution model. The execution model is composed of two phases: a binding process and an instantiation process. The evaluation of views and maps share a common binding process but differ with respect to instantiation. A binding is an environment in which

variables are given values during the instantiation process. Each binding instance provides a set of values to be assigned to the variables. The relationship between bindings and the source data is defined in subsequent clauses of this specification.

8.4 Implementation Environment

The EXPRESS-X language does not describe an implementation environment. In particular, EXPRESS-X does not specify:

- how references to names are resolved;
- how other schemas, schema views, or schema maps are known;
- how input and output data sets are specified;
- how mappings are executed for instances that do not conform to an EXPRESS schema.

The evaluation of a view (i.e. the application of the view to a source data set) produces a view extent. Evaluation of a map may produce entity instances in the target data set. EXPRESS-X does not specify what effect modification of source data may have on views and maps after their evaluation.

9. Declarations

9.1 Overview

This clause defines the various declarations available in EXPRESS-X. An EXPRESS-X declaration creates a new EXPRESS-X item and associates an identifier with it. The item may be referenced elsewhere by this identifier.

EXPRESS-X provides the following declarations:

- View;
- Map;
- Schema_view;
- Schema_map;
- Type_map.

In addition, an EXPRESS-X specification may contain the following declarations defined in ISO10303-11:1994:

– Constant;

– Function;

– Procedure;

– Rule.

9.2 Binding extent declaration

9.2.1 Declaration of qualified binding extents

Syntax:

```
39 binding_decl = [ from_clause ] [ where_clause ] [ identified_by_clause
   ].
```

A qualified binding extent is defined by identification and selection of binding instances.

The FROM language element defines the structure of instances in the binding extent. The FROM language element consists of one or more source_parameter. Each source parameter associates identifiers with an extent.

Syntax:

```
55 from_clause = FROM source_parameter { ';' source_parameter } ';'.
56 source_parameter = source_parameter_id {',' source_parameter_id } ':'
   extent_reference.
```

Rules and restrictions:

a) source_parameter_ids shall be unique within the scope of the map or view declaration.

The binding extent is computed as the cartesian product of instances in the extents referenced in the FROM language element.

EXAMPLE 4 — A binding extent is constructed over the entity extents of entity types item and person.

```
SCHEMA example;
ENTITY item;
  item_number : INTEGER;
END_ENTITY;
ENTITY person;
  name : STRING;
END_ENTITY;
END_SCHEMA;
```

```
VIEW items_and_persons
FROM i : item; p : person;
SELECT
 item_number : INTEGER := i.part_number;
 responsible : STRING := p.name;
END_VIEW;
```

Given a population (written as ISO 10303-21 entity instances):

```
#1=item(123);
#2=item(234);
#33=person('Jones');
#44=person('Smith');
```

the corresponding binding extent is: <#1,#33>,<#1,#44>,<#2,#33>,<#2,#44>.

The WHERE language element defines a selection criteria on binding instances. The WHERE language element, together with the source extents identified in the FROM language element define the qualified binding extent. A binding instance in the binding extent is a member of the qualified binding extent unless one or more domain rule expressions of the WHERE language element evaluates to FALSE for the application of that expression to the binding instance.

The syntax of the WHERE language element is as defined in ISO 10303-11;1994, clause 9.2.2.2.

EXAMPLE 5 — The following example extends the VIEW declaration of Example12 by an WHERE language element to filter specific persons and to join items and persons.

```
VIEW items_and_persons;
FROM i : item; p : person;
WHERE (p.name = 'smith') OR (p.name = 'jones');
      (i.approved_by = p.name);
SELECT
   name : STRING := p.name;
END_VIEW;
```

After the evaluation of the WHERE language element predicates, the output stream will be modified as follows: all grey boxes will be filtered out.

Binding Instance	item				item_version			ddid		person	
		id	version	approved_by		id	ddid		id		name
<#1,#3,#5,#6>	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#6	smith
<#1,#3,#5,#7>	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#7	jones
<#1,#3,#5,#8>	#1	i_1	#3	smith	#3	iv_1	#5	#5	ddid_1	#8	miller
<#2,#4,#5,#6>	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#6	smith
<#2,#4,#5,#7>	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#7	jones
<#2,#4,#5,#8>	#2	i_2	#4	jones	#4	iv_2		#5	ddid_1	#8	miller

9.2.2 Identification of view and target instances

The IDENTIFIED_BY declaration defines an equivalence relation between instances in a qualified binding extent.

Two qualified binding instances are in the same equivalence class if, for each element of the IDENTIFIED_BY clause, evaluating the expression in the context of each of those instances produces result that are instance equal (ISO 10303-11;1994 clause 12.2.2) [NOTE we should really reference our own extended instance equality here].

EXAMPLE 6 — This example illustrates the use of IDENTIFIED_BY.

```
VIEW department;
  FROM e : employee;
  IDENTIFIED_BY e.department_name;
   SELECT
       name : STRING := e.department_name;
END_VIEW;
ENTITY employee;
  name : STRING;
  department_name : STRING;
END_ENTITY;
...
END_VIEW;
```

```
#1=employee('Jones','Engineering');
#2=employee('Smith','Sales');
#3=employee('Doe','Engineering');
```

Given the view and population above, there are two equivalence classes: {#1,#3} and {#2}.

Syntax:

```
57  identified_by_clause = IDENTIFIED_BY expression { ',' expression } ';'.
```

Rules and restrictions:

a) An `expression` in an IDENTIFIED_BY language element shall not refer, through any level of indirection, to the targets of the map or any of their attributes.

9.3 View declaration

9.3.1 Overview

A view declaration creates a view data type and declares an identifier to refer to it.

EXAMPLE 7 — The following view collects the information about persons serving in roles within organizations. This information is collected from two instances of person_and_organization and cc_design_person_and_organization_assignment. The two instances shall be related via the assigned_person_and_organization attribute of the cc_design_person_and_organization_assignment.

```
VIEW arm_person_role_in_organization;
FROM pao : person_and_organization;
     ccdpaoa : cc_design_person_and_organization_assignment;
WHERE ccdpaoa.assigned_person_and_organization :=: pao;
SELECT
  person : person := pao.the_person;
  org : organization := pao.the_organizaion;
  role : label := ccdpaoa.role.name;
END_VIEW;
```

Syntax:

```
120  view_decl = VIEW view_id [ : base_type ][ supertype_rule ] [
     subtype_of_clause ] ';' ( view_partitions | view_decl_body ) END_VIEW
     ';' .
124  view_partition = PARTITION partition_id ';' view_decl_body .
121  view_decl_body = binding_decl view_project_clause .
```

Rules and restrictions:

a) If in a view_decl a subtype_of_clause is specified, no from_clause shall be declared in the `view_decl_body`s of any partition.

b) If no subtype_of_clause is specified, the from_clause is required in every view_decl_body of

that view_decl.

c) Only a return view, clause9.3.5, shall specify a base_type in view_decl.

9.3.2 View attributes

An attribute of a view data type represents a property of the view whose value is computed as the evaluation of its `view_attr_assgnmt_expr`, an expression.

The name of a view attribute (`view_attribute_id`) represents the role played by it associated value in the context of the view in which it appears.

The expression represented by a `view_attr_assgnmt_expr` is evaluated in the context of a qualified binding instance in the qualified binding extent.

If an equivalence class defined by an IDENTIFIED_BY language element contains more than one qualified binding instance, then the value of the view_attr_assgnmt_expression is computed as follows:

– If for each such binding, the evaluation of the `view_attr_assgnmt_expr` (or `map_attr_assgnmt_expr` in the case of a `MAP`) of the attribute produces an equal value, that value is assigned to the attribute.

– If for two or more bindings, the evaluation of the `view_attr_assgnmt_expr` (or `map_attr_assgnmt_expr` in the case of a `MAP`) of the attribute produces unequal values, the indeterminate value is assigned to the attribute.

EXAMPLE 8 — This example illustrates the assignment of values where an equivalence class contain more than one qualified binding instance.

```
(* source schema *)          (* target schema *)
SCHEMA src;                  SCHEMA tar;
ENTITY employee;             ENTITY department;
  name : STRING;               employee : STRING;
  manager : STRING;            manager : STRING;
  dept : STRING;               name : STRING;
END_ENTITY;                  END_ENTITY;
END_SCHEMA;                    END_SCHEMA;
```

```
(* mapping schema *)
SCHEMA_MAP;
SOURCE src; TARGET tar;
MAP department_map AS d : department
FROM e : src.employee
IDENTIFIED_BY e.dept;
SELECT
  d.name := e.dept;
  d.manager := e.manager;
  d.employee := e.name;
END_MAP;
END_SCHEMA_MAP;
```

```
#1=employee('Smith','Jones','Marketing');
#2=employee('Doe','Jones','Marketing');
```

Given the data above the target data set contains one entity instance, #1=department(?,'Jones','Marketing'). The attribute department.employee is indeterminate because the expression for this attribute evaluates to two different values ('Smith' and 'Doe').

Syntax:

```
126  view_project_clause = ( SELECT view_attr_decl_stmt_list) | ( RETURN
     expression ) .
114  view_attr_decl_stmt_list = view_attribute_decl { view_attribute_decl }
     .
115  view_attribute_decl = view_attribute_id ':' [ source_schema_ref '.' ]
     base_type ':=' view_attr_assgnmt_expr ';' .
113  view_attr_assgnmt_expr = expression | choice_expr | inline_view_decl |
     view_call .
```

Rules and restrictions:

a) The `view_attr_assgnmt_expr` shall be assignment compatible with the data type of the view attribute.

b) Each view_attribute_id declared in the view declaration shall be unique within that declaration.

9.3.3 View partitions

A view extent may be partitioned. The extent of a view that is partitioned is the concatenation of the extents defined by its partitions, each partition defining its own FROM language element and selection criteria. Partitions, if present, shall be named. A `partition_id` names a partition.

EXAMPLE 9 — In ISO 10303-201, the application object `organization` may be mapped to either a `person`, an `organization`, or both a `person_and_organization` entity in the AIM. This is specified in EXPRESS-X as follows:

```
VIEW arm_organization
PARTITION a_single_person;
  FROM p : person;
    ...
PARTITION a_single_organization;
  FROM o: organization;
    ...
PARTITION a_person_in_an_organization;
  FROM po: person_and_organization;
     ...
END_VIEW;
```

Syntax:

```
124 view_partition = PARTITION partition_id ';' view_decl_body .
```

Rules and restrictions:

a) All partitions of a `VIEW` declaration shall define the same attributes (including names and types)

b) The attributes of a `VIEW` declaration shall appear in the same order in each of its partitions.

9.3.4 Constant partitions

A partition that omits the FROM, WHERE, and IDENTIFIED_BY clauses is called a constant partition. Such a partition represents a single view instance in the result with no correspondence to the source data.

EXAMPLE 10 — This example illustrates the use of constant partitions.

```
VIEW person;
PARTITION mary;
  SELECT
    name : STRING := 'Mary';
    age : INTEGER := 22;
PARTITION john;
    name : STRING := 'John';
    age : INTEGER := 23;
END_VIEW;
```

9.3.5 Return Views

A view whose body begins with the RETURN keyword in its body computes a BAG of values. One value is computed for each instance of the qualified binding extent by evaluating the expression following the RETURN keyword. A return view does not define a new type. All of the values computed must be mutually type compatible with each other and with the type that optionally may be specified directly after the name of the view.

Rules and restrictions:

a) A return view shall not use the SELECT language element in any partition.

b) A return view shall not specify the subtype_of_clause language element.

EXAMPLE 11 — EXAMPLE. This example defines a bag whose members are instances of the type car that have the value 'red' in their color attribute.

```
VIEW red_car;
  FROM rc:car;
  WHERE rc.color ='red';
  RETURN rc;
END_VIEW;
```

EXAMPLE 12 — EXAMPLE. This example defines a bag whose members are strings. The strings come from two sources.

```
VIEW owner_name : STRING;
  PARTITION one;
    FROM po:person;
    RETURN po.name;
  PARTITION two;
    FROM or: organization;
    RETURN or.name;
END_VIEW;
```

9.3.6 Specifying subtype views

EXPRESS-X allows for the specification of views as subtypes of other views, where a subtype view is a specialization of its supertype. This establishes an inheritance (i.e., subtype/supertype) relationship between the views in which the subtype inherits the properties (i.e., attributes and selection criteria) of its supertype. A view is a subtype view if it contains a SUBTYPE declaration. The extent of a subtype view is a subset of the extent of its supertype as defined by the selection criteria defined by the WHERE language element in the subtype.

A subtype view inherits attributes from its supertype view(s). Inheritance of attributes shall adhere to the rules and restrictions of attribute inheritance defined in ISO 10303-11;1994 clause 9.2.3.3.

A subtype view declaration may redefine attributes found in one of its supertypes. The redefinition of attributes shall adhere to the rules and restrictions of attribute redefinition defined in ISO 10303-11;1994 clause 9.2.3.4.

A view instance shall be created if the selection criteria of the most general supertype is satisified. The view instance shall have the type corresponding to a subtype view if all of the selection criteria conditions in the subtype view in addition to all of its supertype views evaluate to TRUE or UNKNOWN.

Syntax:

```
102 subtype_of_clause = SUBTYPE OF '(' view_or_map_reference  { ','
    view_or_map_reference } ')' .
```

Rules and restrictions:

a) A view declaration shall contain either a FROM language element or a subtype language element, but not both.

b) A subtype view shall not specify the IDENTIFIED_BY lanaguage element.

c) Exactly one supertype view of a subtype view shall define a FROM language element

d) The partitions of a subtype view shall be a subset of the partitions of its supertype view.

e) A subtype view shall not use the return language element.

EXAMPLE 13 — The following view illustrates subtyping. The view `male` defines an additional membership requirement (`gender = 'M'`) for view instances of the subtype.

```
VIEW person;
FROM e:employee;
END_VIEW;

VIEW male SUBTYPE OF (person);
WHERE e.gender = 'M';
...
END_VIEW;
```

EXAMPLE 14 — This example illustrates the use of partitions and subtype views.

```
VIEW j;
PARTITION first:
FROM s:three, t:four
WHERE cond6;

PARTITION second:
FROM r:four, q:five
WHERE cond7;
END_VIEW;
```

```
VIEW k SUBTYPE OF (j);
PARTITION second:
WHERE cond9;
END_VIEW;
```

Any subtype view for which ‘k’ is a supertype can only include partition ‘second’.

9.3.7 SUPERTYPE constraints

A view declaration may define SUPERTYPE constraints (ISO standard 10303 part (11) clause 9.2.4). Whether or not a SUPERTYPE constraint is satisfied has no effect on the execution model or content of view extents.

EXAMPLE 15 —

```
VIEW a ABSTRACT SUPERTYPE OF ONEOF(b ANDOR c, d);
 ...
END_VIEW;
```

An instance of ‘a’ is valid if it has at least two types (‘a’ and something else) because of the ABSTRACT keyword, and one of the other types is either ‘d’ or some combination of ‘b’ and ‘c’ because of the ONEOF keyword.

9.4 Map declaration

9.4.1 Overview

The MAP declaration supports the specification of correspondence between semantically equivalent elements of two or more EXPRESS models possessing differing structure. Each MAP declaration specifies how source schema entity instances of one or more types are to be mapped to target instances.

A map declaration supports, in a single declaration, the mapping from many source entities to many target entities.

Syntax:

```
 67  map_decl = MAP map_id AS target_parameter { target_parameter }  (
     (map_decl_body { map_partitions }) | map_decl_body ) END_MAP ';' .
 71  map_partition = PARTITION partition_id ':' map_decl_body .
 68  map_decl_body = [subtype_of_clause] binding_decl  {
     entity_instantiation_loop }  map_project_clause .
104  target_parameter = [ target_parameter_id { ',' target_parameter_id }
     ':' ] [ AGGREGATE [ bound_spec ] OF ] target_entity_reference ';' .
```

The header identifies one or more entity types defined in the target EXPRESS schema to be created upon evaluation.

A target entity type shall not be mapped in more than one MAP declaration in which the headers of those declarations consist only of a single target entity type. However, one target entity can be mapped in more than one MAP declarations (say n), if n-1 MAP declarations are network mappings. The MAP declaration is named (map_id).

NOTE — A single target entity type may be mapped in various ways by means of partitions.

EXAMPLE 16 — In the example below, a pump in the source data model is mapped to a product and product_related_product_category.

```
MAP network_for_pump AS pr : product;
                        prpc : product_related_product_category;
FROM p : pump
  pr.id := p.id;
  pr.name := p.name;
  prpc.name := 'pump';
  prpc.products := [ pr ];
END_MAP;
```

The initial values of the attributes of the newly created instance(s) are indeterminate.

9.4.2 Evaluation of the MAP body

The MAP declaration may define a SELECT language element consisting of either an extent reference or a number of target attribute assignment statements. Alternatively, the SELECT language element may be omitted entirely:

- A SELECT language element specifying a source_parameter_id signifies that the instances of the corresponding extent are to be mapped identically. (i.e. such that they are value equal to instances in the source extents).
- A SELECT language element specifying target attribute assignment statements (map_attribute_declarations) is used to assign values to the attributes of the target entity instances.
- If the SELECT language element is omitted, entity instances value equal to those specified in the FROM language element are created on evaluation.

Syntax:

```
73  map_project_clause = (SELECT map_attr_decl_stmt_list) | ( RETURN
    expression ) .
64  map_attr_decl_stmt_list = map_attribute_declaration {
    map_attribute_declaration } .
65  map_attribute_declaration = [ target_parameter_ref [ index_qualifer ]
    [ group_qualifier ] '.' ] attribute_ref [ index_qualifier ] ':='
    map_attr_assgnmt_expr ';' .
```

The `map_attr_assgnmt_expr` shall produce a value that is assignment compatible with the target entity attribute (see ISO10303-11;1994 clause 13.3).

The syntactic form:
`SELECT source_parameter_id`
declares that an entity instance value equivalent to that bound to `source_parameter_id` shall appear in the target data set.

The syntactic form
`SELECT map_attr_decl_stmt_list`
assigns values (`map_attr_assgnmt_expr`) to the target entity attributes (l-values) identified by the syntactic form of the left-hand side of `map_attribute_declaration` (i.e., before the `':='`).

A `map_attr_decl_stmt_list` may assign to the multiple elements of an aggregate of a target entity type. The order of execution of the attribute assignments in this case is arbitrary.

9.4.3 Instantiation of aggregates

Evaluation of a map may produce aggregates of target entity types. The declared type returned in this situation is AGGREGATE. The initial value of the aggregate is indeterminant.

Syntax:

```
104  target_parameter = [ target_parameter_id { ',' target_parameter_id }
     ':' ] [ AGGREGATE [ bound_spec ] OF ] target_entity_reference ';' .
```

Rules and restrictions:

a) If `bound_spec` is specified it is treated as a constraint.

EXAMPLE 17 — Body of a MAP declaration with attribute assignments of multiple target instances of the same entity.

```
MAP connection_zone_shapes AS
    pdr : AGGREGATE OF aim.property_definition_representation;
    sr  : AGGREGATE OF aim.shape_representation;
  FROM cz : arm.connection_zone;
  FOR EACH shape IN cz.zone_shape INDEXING i;
  SELECT
    sr[i].name := 'zone shape';
    pdr[i].definition := pd@connection_zone_map(cz);
    pdr[i].used_representation := sr[i];
END_MAP;
```

9.4.4 Iteration under a single binding instance

9.4.4.1 Overview

The instantiation_loop_control and repeat_control provides two mutually exclusive forms of iteration: iteration over the collection of instances in an EXPRESS aggregate; and interaction incrementing a numeric variable. The latter of these, provided by repeat_control is described in ISO 10303-11; 1994.;

Syntax:

```
46 entity_instantiation_loop = FOR instantiation_loop_control ';' .
62 instantiation_loop_control = instantiation_foreach_control |
   repeat_control .
61 instantiation_foreach_control = EACH variable_id IN
   source_attribute_reference [ INDEXING variable_id ] { AND variable_id
   IN source_attribute_reference [ INDEXING variable_id ] } .
```

Rules and restrictions:

a) variable_id after the keyword EACH is of the same type as the elements of source_attribute_reference.

b) variable_id after the keyword INDEXING is of type NUMBER with values greater than one.

9.4.4.2 Control by numeric increment

The FOR repeat control allows for the iteration under a single binding instance by means of the EXPRESS repeat_control.

EXAMPLE 18 — This example illustrates the use of the EXPRESS `repeat_control` in Express-X target instantiation. A collection of target child entity instances are created for each source parent entity. The number created is specified by the parent entity attribute number_of_children.

```
SCHEMA source;
ENTITY parent;
number_of_children : INTEGER;
END_ENTITY;
END_SCHEMA;
```

```
SCHEMA target;
ENTITY parent;
END_ENTITY;
ENTITY child;
   parent : parent;
END_ENTITY;
END_SCHEMA;
```

```
SCHEMA_MAP example;
  SOURCE src : source;
  TARGET tar : target;
MAP tp AS tar.parent;
FROM sp : src.parent;
END_MAP;
MAP children_map AS c : AGGREGATE [0:?] OF tar.child;
FROM p : src.parent;
FOR i := 1 TO p.number_of_children
SELECT
  c[i].parent := p;
END_MAP;
END_SCHEMA_MAP;
```

EXAMPLE 19 — We assume that for each source instance of item exactly three corresponding target instances have to be generated. That is specified in the following mapping specification.

```
ENTITY item_with_duplicates;
  id : STRING;
  index : INTEGER;
END_ENTITY;
```

```
MAP iwd AS AGGREGATE [3:3] OF item_with_duplicates
FROM i : item
  FOR var := 1 TO 3
   SELECT
    id := i.id;
    index := var;
END_MAP;
```

	item_with_duplicates									
		id			index					
	item					item_version			ddid	
		id	its_version	approved_by			id	its_ddid		id
0x01	#1	i_1	#3	smith	1	#3	iv_1	#5	#5	ddid_1
0x02	#1	i_1	#3	smith	2	#3	iv_1	#5	#5	ddid_1
0x03	#1	i_1	#3	smith	3	#3	iv_1	#5	#5	ddid_1
0x04	#2	i_2	#4	jones	1	#4	iv_2		#5	ddid_1
0x05	#2	i_2	#4	jones	2	#4	iv_2		#5	ddid_1
0x06	#2	i_2	#4	jones	3	#4	iv_2		#5	ddid_1

9.4.4.3 Control by iteration over an aggregate

Under the `instantiation_foreach_control`, at each iteration step, the next element of the source attribute is bound to a variable and optionally the index position of that element is bound to an iterator variable. The scope of these variable bindings includes the `map_project_clause`. For example, for each element of the source attribute of type aggregate a target instance can be generated and the element value can be assigned to a corresponding target attribute of type.

EXAMPLE 20 — In the following example, all item versions of one item are grouped together in the source data model. In contrast, each item version is a stand-alone instance in the target data model. This example shows that the FOR loop specifies an iteration over the elements of the source attribute `item_with_versions.id_of_versions`. For each source instance and for each element in that attribute a target instance is created. The target attribute `item_id` is mapped in the same way for all the target instances which of `item_version` which correspond to the same underlying `item_with_versions`. The target attribute `version_id` is assigned to the value of the iterator variable `version_iterator`.

```
ENTITY item_version; --target data model
  item_id    : STRING;
  version_id : STRING;
END_ENTITY;

ENTITY item_with_versions; -- source data model
  id             : STRING;
  id_of_versions : LIST OF STRING;
END_ENTITY;
```

```
MAP iv : AGGREGATE [0:?] OF item_version
FROM iwv : item_with_versions;
FOR EACH version_iterator OF iwv.id_of_versions INDEXING i
SELECT
  iv[i].item_id    := iwv.id;
  iv[i].version_id := version_iterator;
END_MAP;
```

For example, the following target instances are built from the source instance below.
Source instance set:

```
#1 = item_with_versions(1,(10,11,12));
```

Target instance set:

```
#1 = item_version(1,10);
#2 = item_version(1,11);
#3 = item_version(1,12);
```

9.4.5 Partitions within a MAP declaration

A single target entity may be related in a specific way to source data for some instances and differently to source data for some other instances. Map partitions may be used to specify these differing relations. A MAP declaration may be partitioned, each partition defining its own FROM language element and selection criteria. Partitions, if present, shall be named. A `partition_id` names a partition.

If multiple target entities are listed in the header of the `MAP` declaration, different subsets of those entities may be created by each partition.

Syntax:

```
71  map_partition = PARTITION partition_id ':' map_decl_body .
```

Rules and restrictions:

a) The `partition_id` shall be unique with respect to the inheritance hierarchy of the corresponding target entity.

b) For every target entity declared in the map header, at least one partition shall be defined to create instances for it.

EXAMPLE 21 — This example illustrates how various source entity types may be mapped into a single target entity type using a MAP declaration containing partitions.

```
(*  source schema *)            (* target schema *)
SCHEMA src;                     SCHEMA tar;
ENTITY student;                 ENTITY person;
     name : STRING;               name : STRING;
END_ENTITY;                     END_ENTITY;
ENTITY employee;                END_SCHEMA;
  name : STRING;
END_ENTITY;
END_SCHEMA;

(* mapping schema *)
SCHEMA_MAP example;
MAP student_employee_to_person AS p : tar.person;
PARTITION student;
FROM s : src.student;
SELECT
  p.name  := s.name;
PARTITION employee;
FROM e : src.employee;
SELECT
  p.name := e.name;
END_MAP;
```

9.4.6 Mapping to an entity type and its subtypes

EXPRESS-X allows for the specification of a map as a subtype of another map. Subtype map declarations may extend the collection of entity instances created by its supertype map, specialize those instances created and require additional selection criteria beyond those specified in the supertype map. The specification of a target attribute assignment declared in a supertype map is inherited by its subtype maps. Through this means the pattern of inheritance present in the target schema can be duplicated in the mapping declarations.

Whether a subtype map extends the collection of entity instances created by its supertype map or specializes those instance created depends on whether the subtype map references target_parameter_ids declared in the supertype map or whether it declare its own target_parameter_ids:

– If a map's selection criteria and that of all its supertype maps is satisfied, the map may execute.

– A subtype map may reference in its `map_decl_header` a target parameter that is declared in any of its supertype maps. The type created is the composition of types identified by the subtype map target parameter and all supertype maps declaring a target parameter with this target parameter id.

– A subtype map may introduce a target_parameter_id that is not defined in any of the supertype

maps. In this case a new target entity of the type defined by the target parameter is created.

– Rules and Restrictions:

– The type combination must be one that is valid in the target schema.

EXAMPLE 22 — A mapping schema illustrating the assignment to attributes declared in supertypes and subtypes through supertype and subtype maps. Source entities are of one type, `s_project`. Target entities are of type `t_project` and perhaps one of its subtypes, `in_house_project` and `external_project`. The target_parameter_id, tp, used in the supertype map (project_map) is used again in its subtype maps (in_house_map, ext_map) signifying that the corresponding target entity is specialized in the subtype maps.

```
SCHEMA source_schema;
ENTITY s_project;
  name : STRING;
  project_type : STRING;
  cost : INTEGER;
  price : INTEGER;
  vendor : STRING;
END_SCHEMA;

SCHEMA target_schema;
ENTITY t_project;
SUPERTYPE OF (ONEOF (in_house_project, external_project));
  name : STRING;
  cost : INTEGER;
  management : STRING;
END_ENTITY;
ENTITY in_house_project;
SUBTYPE OF (t_project);
END_ENTITY;
ENTITY external_project;
SUBTYPE OF (t_project)
  price : INTEGER;
END_ENTITY;
END_SCHEMA;
```

```
MAP project_map AS tp : target_schema.t_project;
FROM p : source_schema.s_project;
SELECT
     tp.name := p.name;
     tp.cost := p.cost;
END_MAP;

MAP in_house_map AS tp : target_schema.in_house_project;
SUBTYPE OF project_map;
WHERE (p.project_type = 'in house');
SELECT
  tp.management := CHOICE (cost < 50000) THEN 'small accts'
                   ELSE 'large accts' ENDIF;
END_MAP;

MAP ext_map AS tp : target_schema.external_project;
SUBTYPE OF project_map;
WHERE (p.project_type = 'external');
SELECT
  tp.price := p.price;
  tp.management = p.vendor;
END_MAP;
```

9.4.7 Explicit declaration of complex entity data types

Complex entity data types (see ISO10303-11:1994, clause3.2.1) may be explicitly declared in the map header. A complex entity data type is referenced by an expression that lists the partial complex entity data types that are combined to form it, separated by '&'.

The partial complex entity data types may be listed in any order.

Any partial complex entity data types that are included in another partial complex entity data type via inheritance are not listed.

Syntax:

```
44 complex_entity_spec = entity_reference '&' entity_reference { '&'
   entity_reference } .
```

Rules and restrictions:

a) Each entity_ref shall be a reference to an entity which is visible in the current scope.

b) The referenced complex entity data type shall describe a valid domain within some schema (see ISO10303-11:1994, annexB).

c) A given entity_ref shall occur at most once within a complex_entity_ref.

d) For each entity_reference declared in the complex_entity_spec, none of its supertype shall be declared.

9.5 Schema_view declaration

A schema_view declaration defines a common scope for a collection of related mapping declarations. A schema_view may contain the following kinds of declarations:

– constant declaration ();

– function declaration (clause9.6);

– procedure declaration (clause 9.7);

– rule declarations (clause 9.11);

– view declaration (clause 9.3).

The order in which declarations appear within a schema_view declaration is not significant.

Declarations in one schema_view or EXPRESS schema may be made visible within the scope of another schema_view via an interface specification as described in clause 13.

Syntax:

```
95  schema_view_decl = SCHEMA_VIEW schema_view_id {
    reference_clause_extended } [ constant_decl ]
    schema_view_body_element_list END_SCHEMA_VIEW ';' .
82  reference_clause_extended = REFERENCE FROM foreign_ref [ '('
    resource_or_rename { ',' resource_or_rename } ')' ] ';' .
93  schema_view_body_element = function_decl | procedure_decl | view_decl
    .
```

EXAMPLE 23 — ap203_arm names a schema_view that may contain declarations defining a view over the schema config_control_design in terms of the domain expert's understanding of the information requirements.

```
SCHEMA_VIEW ap203_arm;
REFERENCE FROM config_control_design;
VIEW part_version ...
(* other declarations as appropriate *)
END_SCHEMA_VIEW;
```

9.6 Schema_map declaration

A schema_map declaration defines a common scope for a collection of related mapping declarations.

EXAMPLE 24 — iges2step names a schema_map that may contain declarations for translating geometry defined using an EXPRESS model base upon IGES into a model based on ISO 10303-203.

```
SCHEMA_MAP iges2step;
TARGET step_schema;
SOURCE iges_express_schema;
MAP iges_structure ...
(* other declarations as appropriate *)
END_SCHEMA_MAP;
```

The order in which declarations appear within a schema_map declaration is not significant. In particular, the order of the declarations has no effect upon the resulting mapping.

Declarations in one schema_map may be made visible within the scope of another schema_map via an interface specification as described in clause13.3.3

A schema_map may contain the following kinds of declarations:

- constant declaration (clause9.5);
- function declaration (clause9.6);
- procedure declaration (clause9.7);
- type_map declaration (clause9.8);
- view declaration (clause9.3);
- map declaration (clause9.4);
- rule declaration (clause 9.11).

Syntax:

```
 87  schema_map_decl = SCHEMA_MAP schema_map_id target_interface_spec {
     target_interface_spec } source_interface_spec { source_interface_spec
     } { map_interface_spec } { type_mapping_stmt } [ constant_decl ]
     schema_map_body_element_list END_SCHEMA_MAP ';' .
108  target_interface_spec = TARGET schema_ref_or_rename [ REFERENCE
     resource_or_rename { ',' resource_or_rename } ] ';' .
100  source_interface_spec = SOURCE schema_ref_or_rename [ REFERENCE
     resource_or_rename { ',' resource_or_rename } ] ';' .
 69  map_interface_spec = IMPORT_MAPPING schema_map_or_view_ref_or_rename [
     REFERENCE resource_or_rename { ',' resource_or_rename } ] ';' .
111  type_mapping_stmt = TYPE_MAP type_reference FROM type_reference ';'
     type_map_stmt_body type_map_stmt_body   END_TYPE_MAP ';' .
 85  schema_map_body_element = function_decl | procedure_decl | view_decl |
     map_decl | create_map_decl .
```

The body of a `schema_map` shall have the same form as the body of a `schema` in ISO 10303-11;1994, with the following exceptions:

– The `schema_map` shall include at least one `MAP` declaration.

– The `schema_map` shall include a `target_interface_spec` declaration.

– The `schema_map` shall include a `source_interface_spec` declaration.

– The `schema_map` shall not include the `interface_specification` declaration (defined in ISO 10303-11;1994).

– The `schema_map` shall not include the `entity` declaration (defined in ISO 10303-11;1994).

– The `schema_map` shall not include the `type` declaration (defined in ISO 10303-11;1994).

EXAMPLE 25 — This example illustrates the use of required EXPRESS-X declarations. `t1`, `t2`, `t3`, `s1` and `s2` designate EXPRESS schema. `other_map` designates an EXPRESS-X schema.

```
SCHEMA_MAP map_name;
  TARGET t1, t2, t3;
  SOURCE s1, s2;
IMPORT MAPPING other_map;
END_SCHEMA_MAP;
```

9.7 Create declaration

The `CREATE` declaration defines the form of an entity that, subject to a logical expression, shall be created in the target data set. The `logical_expression` is evaluated against entity extents identified in the `target_entity_reference`. If the `logical_expression` does not evaluates to

FALSE or if no `logical_expression` is specified, an entity shall be created in the target data set. If the logical_expression evaluates to the indeterminate value, the behaviour is undefined.

Syntax:

```
45 create_map_decl = CREATE instance_id ':' target_entity_reference ';' [
   WHERE logical_expression ';' ] map_attr_decl_stmt_list END_CREATE ';'
   .
```

Rules and restrictions:

a) The `CREATE` declaration shall not be used except within the scope of a schema map.

b) `logical_expression` shall evaluate to either a `LOGICAL` value or indeterminate.

c) `target_entity_reference` shall refer to entity identifiers defined in a target schema.

d) Attribute references of the `map_attr_decl_stmt_list` shall refer to attributes of entities identified in the `target_entity_reference`.

EXAMPLE 26 — In the following, an instance of application_context is created in the target data set provided that the entity extent of `item` (an entity type in a source schema) contains at least one instance.

```
CREATE APPCNT INSTANCE_OF application_context
WHERE SIZEOF(EXTENT(item)) > 0;
application := '';
END_CREATE;
```

9.8 Constant declaration

Constants may be defined for use within the WHERE language element of a view or map declaration, or within the body of a map declaration or algorithm.

Constant declarations are as defined in ISO10303-11:1994 clause9.4.

9.9 Function declaration

Functions may be defined for use within the WHERE language element of a view or map declaration, or within the body of a map declaration.

Function declarations are as defined in ISO10303-11:1994 clause9.5.1.

9.10 Procedure declaration

Procedures may be defined for use within the body of a map declaration.

Procedure declarations are as defined in ISO10303-11:1994 clause9.5.2.

9.11 Rule declaration

Rules may be defined for use within the SCHEMA_VIEW and SCHEMA_MAP language element.

Rule declarations are as defined in ISO 10303-11:1994 clause 9.6.

9.12 Type map declaration

A type map declaration specifies how a value of a defined type is mapped to a value of another type within the scope of a schema map.

Syntax:

```
111  type_mapping_stmt = TYPE_MAP type_reference FROM type_reference ';'
     type_map_stmt_body type_map_stmt_body   END_TYPE_MAP ';' .
110  type_map_stmt_body = [ schema_ref '.' ] base_type ':='
     type_assgnmt_expr ';' .
```

EXAMPLE 27 — The following specifies the mapping between the types `dollar` and `dmark`. The target type `dmark` is mapped to the source type `dollar` by multiplying `dollar` with the factor `1.5` to derive `dmark`. Any attribute assignment where a target attribute of type `dmark` is mapped an expression of type dollar, the first type_map_stmt_body is applied.

```
TYPE_MAP dmark FROM dollar;
  dmark := 1.5 * dollar;
  dollar := dmark / 1.5;
END_TYPE_MAP;
```

The mapping is applied whenever the map_attr_expression evaluates to one of the base_types declared in the type_map_stmt_body and the map attribute is declared to be of the other type_map_stmt_body base_types. The appropriate type_map_stmt_body is applied to the value of the map attribute_expr and the resulting value is assigned.

Rules and restrictions:

a) `base_type` shall a defined data type.

10. Expressions

10.1 Overview

Expressions are combinations of operators, operands, and function calls that are evaluated to produce a value.

Precedence of operators and the order of evaluation of expressions are as defined in ISO 10303-11:1994 clause12. [POD replace this with the table, extended with @].

Entity constructors create instances that are local only to the function or procedure and do not exist in either the target or the source.

10.2 View call

A view call is an expression that evaluates to a view instance or aggregate of view instances. The view call provides a means to access a view instance through arguments corresponding to its binding instance (when no IDENTIFIED_BY is defined) or IDENTIFIED_BY language element expressions (when IDENTIFIED_BY is defined). If no view instance corresponds, the call evaluates to indeterminate. A view call identifies a single partition of a view; if the view contains more than one partition, a partition_qualification shall be present. When no IDENTIFIED_BY language element is present in the partition, the number, type, and order of the actual parameters shall agree with that of the source parameters of the FROM language element in the partition. When an IDENTIFIED_BY language element is present, the number, type and order of the actual parameters shall agree with that of the expressions of the IDENTIFIED_BY langauge element.

A view call referencing a constant partition shall be passed an empty parameter list.

Syntax:

```
119  view_call = view_reference [ partition_qualification ] '(' expression {
     ',' expression } ')' .
```

EXAMPLE 28 — This example illustrates the use of a view call to define a relationship between two view data types. The IDENTIFIED_BY language element in the person_view specifies one expression, `a.creator`; view calls to person_view will therefore be supplied with one argument, a STRING which is also the creator attribute of an approval entity instance. The IDENTIFIED_BY clause in this view also serves to ensure the uniqueness of person_view instances (i.e. no two view instances will have the same name attribute).

```
SCHEMA_VIEW example;
VIEW approver
 PARTITION person_part;
  FROM a : approval; p : person;
  WHERE a.creator = p.name;
  IDENTIFIED_BY a.creator;
  SELECT
    approver_id : INTEGER := p.id;
 PARTITION org_part;
  FROM a : approval; o : organization;
  WHERE a.creator = o.name;
  IDENTIFIED_BY a.creator;
  SELECT
    approver_id : INTEGER := o.id;
END_VIEW;

VIEW design_order;
  FROM a : approval;
  SELECT
    id : STRING := a.id;
    approved_by : approver :=
                  approver\person_part(a.creator);
END_VIEW;
END_SCHEMA_VIEW;

SCHEMA src_schema;
ENTITY approval;
  id : STRING;
  creator : STRING;
END_ENTITY;
ENTITY person;
  name : STRING;
  id : INTEGER;
END_ENTITY;
END_SCHEMA;
(* Source data set in ISO 10303-21 form *)
#1=approval('a_1','Jones');
#2=approval('a_2','Smith');
#3=approval('a_3','Jones');
#4=person('Jones',123);
#5=person('Smith',234);
```

```
(* Resulting view instances in ISO 10303-21 form *)
#101=approver(123);
#102=approver(234);
#103=design_order('a_1',#101);
#104=design_order('a_2',#102);
#105=design_order('a_3',#101);
```

If one or more of the actual parameters is indeterminate, the result of the view call is a SET containing those view instances of the view extent that correspond to the non-indeterminate parameter values provided. If no view instances correspond the view call evaluates to indeterminate.

EXAMPLE 29 — In the following, the various versions associated with a part are collected by using a partial explicit binding. Returned by the explicit binding call version_and_its_product is the subset of the extent for which the second component of the binding is equal to the specified product instance.

```
VIEW part;
FROM (p : product)
SELECT
  versions : SET OF version_and_its_product
     := version_and_its_product(?, p);
END_VIEW;
```

10.3 Map Call

A map call is an expression that evaluates to a target entity instance. A map call identifies a single partition of a map; if the map contains more than one partition, a partition_qualification shall be present. When no IDENTIFIED_BY language element is present in the partition, the number, type, and order of the actual parameters shall agree with that of the source parameters of the FROM language element in the partition. When an IDENTIFIED_BY language element is present, the number, type and order of the actual parameters shall agree with that of the expressions of the IDENTIFIED_BY langauge element.If the view call references a constant partition, then a empty parameter list shall be passed.

Syntax:

```
66 map_call = target_parameter_ref [ map_or_partition_qualification ] '('
   expression { ',' expression } ')' .
70 map_or_partition_qualification = '@' map_ref | | '@'  map_ref '.'
   partition_ref .
```

Rules and restrictions:

a) `target_parameter_ref` shall refer to a parameter reference declared in the MAP referenced as `map_ref`.

EXAMPLE 30 — This example illustrates the use of a map call to define a relationship between entities in the target schema.

```
(* source schema *)
SCHEMA source;
ENTITY approval;
  id : STRING;
  creator : STRING;
END_ENTITY;
END_SCHEMA;

SCHEMA target;
ENTITY person;
  id : STRING;
END_ENTITY;

ENTITY design_order;
  id : STRING;
  approved_by : person;
END_ENTITY;

MAP_SCHEMA example;
MAP person_map AS p : target.person;
FROM a : approval
IDENTIFIED_BY a.creator
SELECT
  p.id := a.creator;
END_MAP;

MAP design_order_map AS d : target.design_order;
FROM a : approval
SELECT
  d.id := a.id;
  d.approved_by := p@person(a.creator); -- explicit binding
END_MAP;
END_MAP_SCHEMA;

(* source instance set written as ISO 10303-21 instances *)
#1 = approval('a_1','miller');
#2 = approval('a_2','jones');
#3 = approval('a_3','miller');

(* Resulting target instances in ISO 10303-21 form *)
#101=person('Jones');
#102=person('Smith');
#103=design_order('a_1',#101);
#104=design_order('a_2',#102);
#105=design_order('a_3',#101);
```

A partial explicit binding is an explicit binding in which one or more of the parameters is indeterminate. The result of a partial explicit binding is the subset of the extent that matches the parameter values that are provided. If the subset is empty, the result of the partial explicit binding shall be indeterminate.

EXAMPLE 31 — In the following, the various versions associated with a part are collected by using a partial explicit binding. Returned by the explicit binding call version_and_its_product is the subset of the extent for which the second component of the binding is equal to the specified product instance.

```
VIEW part;
FROM (p : product)
SELECT
  versions : SET OF version_and_its_product
     := version_and_its_product(?, p);
END_VIEW;
```

```
VIEW version_and_its_product;
FROM (pdf : product_definition_formation, p : product)
WHERE p :=: pdf.of_product;
SELECT
  the_version : product_definition_formation := pdf;
END_VIEW;
```

10.4 FOR expression

The FOR expression collects the result of iteration of an expression over the elements of an EXPRESS aggregate. The collection is returned as an EXPRESS aggregate. The FOR expression may be used in the map_attr_assgnmnt_expr.

The FOR expression iteration mechanism allows each element to be evaluated against a selection criteria. Elements and of the aggregate can be processed step by step, selected, and manipulated.

The FOR expression is introduced for attribute assignment statements of MAP declarations to process a set of elements and to assign a set as a result to the target attribute. For this purpose, an iteration mechanism is used where all elements of the set can be processed step by step, selected, and manipulated.

The iteration of the FOR expression is controlled either by the repeat control known from EXPRESS (cf., ???). Alternatively, a more declarative approach can be specified using the FOR EACH concept. In the latter case, the following language elements are available.

– The EACH language element defines the (name of the) iterator variable. That is, in each processing step of the loop of the FOR expression, an element of the set is assigned to this iterator. The set is determined by the IN- (and the FROM-) language element.

– The IN language element specifies the set over which it has to be iterated over. This is either an (entity) extend. In this case the FROM language element is optional. That is, if it shall be iterated over exact one (entity) extent without further restrictions the FROM language element need not to

be specified. Alternatively, if it shall be iterated over an extent which is built upon many joined source extents, the FROM language element (and the WHERE language element) are needed.

In addition to the entity extent, it can also be iterated over an attribute of type AGGREGATE. In this case, the FROM language element is optional: if the source entity of this attribute to be iterated over is not specified in the FROM language element of the MAP declaration, it shall be specified in the FROM language element of the FOR expression.

– The FROM language element of the FOR expression has the same semantics as the FROM language element of the MAP declaration (cf., ???).

– The WHERE language element of the FOR expression has the same semantics as the WHERE language element of the MAP declaration (cf., ???).

– The RETURN language element specifies an expression which has to be processed for each element during the iteration. All processed elements together build the result aggregate data type which is returned to the target attribute.

EXAMPLE 32 — FOR expression.

```
(* Source schema *)
ENTITY product_definition;
  product_name : STRING;
  description : STRING;
END_ENTITY;

ENTITY product_definition_name;
  name    : STRING;
  of_product_definition : product_definition;
END_ENTITY;

(* Target schema *)
ENTITY component;
  names : SET [0:?] OF STRING;
  product_name  : STRING;
  description  : STRING;
END_ENTITY;
```

In this example, the target entity `component` maps to the source entity `product_definition` and all instances of `product_definition_name` which reference one instance of `product_definition` are grouped into the target attribute `component.names`. This is specified as follows.

```
Mapping definition:
MAP component
FROM pd : product_definition
SELECT
  description := pd.description;
  product_name := pd.product_name;
  names := FOR EACH pdn_instance
                IN pdn
              FROM pdn : product_definition_name
             WHERE pdn.of_product_definition :=: pd
            RETURN pdn_instance.name
END_MAP;
```

This example also shows that the scope of the FROM language element of the MAP declaration can be extended by the FROM language element of an FOR expression within this MAP declaration. That is, `product_definition_name` is not within the scope of the root entity of the FROM language element of the MAP declaration `product_definition`. In this case, the FOR expression specifies the so-called outer join operation. That is, for each instance of `product_definition` a target instance of `component` is built independent of the existence of instances of `product_definition_name` which references this `product_definition`. If such instances of `product_definition_name` do not exist, the value of `component.names` is the empty set. Otherwise, those instances (resp. the value `product_definition_name.name`) are assigned to the attribute `component.names`.

The RETURN language element can be nested in order to map attributes which are of type AGGREGATE OF AGGREGATE. This is shown in the following example.

EXAMPLE 33 — Nested FOR expression. The example 32 is extended as follows.

```
Source schema:
ENTITY product_definition;
  (* as defined in Ex. 32 *)
END_ENTITY;

ENTITY product_definition_name;
  (* as defined in Ex. 32 *)
END_ENTITY;

ENTITY product_definition_value;
  of_pdn : product_definition_name;
  value : STRING;
END_ENTITY;

Target schema:
ENTITY component;
  values : SET [0:?] OF SET [0:?] OF STRING;
  product_name  : STRING;
  description  : STRING;
END_ENTITY;
```

In addition to example 32, all instances of `product_definition_value` which reference one instance of `product_definition_name` are grouped together and are assigned to the inner aggregate of `component.values`. This is specified as follows.

```
Mapping definition:
MAP component
FROM pd : product_definition
SELECT
  description := pd.description;
  product_name := pd.product_name;
  names := FOR EACH pdn_instance
                IN pdn
              FROM pdn : product_definition_name
             WHERE pdn.of_product_definition :=: pd
            RETURN FOR EACH pdv_instance
                        IN pdv
                      FROM pdv : product_definition_value
                     WHERE pdv.of_pdn :=: pdn_instance
                    RETURN pdv_instance.value;
END_MAP;
```

The FOR expression supports parallel iteration (i.e. iteration where two or more iterator variables are assigned to elements of sets). During each step of the iteration loop, all the iterator variables are assigned to the next element of the corresponding set. This is shown in the following example.

EXAMPLE 34 — Parallel iteration with the FOR expression.

```
Source schema:
ENTITY persons;
  firstname : SET [0:?] OF STRING;
  lastname : SET [0:?] OF STRING;
END_ENTITY;
```

```
Target schema:
ENTITY set_of_persons;
  name : SET [0:?] OF STRING;
END_ENTITY;
```

It is assumed that `persons.firstname[i]` corresponds to `persons.lastname[i]` and that those two values have to be concatenated and have to be assigned to `set_of_persons.name[i]`.

```
Mapping specification:
MAP set_of_persons
FROM p : persons
SELECT
  name := FOR EACH firstname_value IN p.firstname AND
              EACH lastname_value  IN p.lastname
            RETURN firstname_value + lastname_value;
END_MAP;
```

This example also shows that the FROM language element of the FOR expression is optional when it is a

subset of the FROM language element of the MAP declaration. In this example, no predicates are needed to select specific elements of the extent which is given by the IN language element. Thus, the WHERE language element is omitted.

If the scope of the extent of the FOR loop (as specified by the `foreach_in_clause_arg` resp. the `repeat_control`) is empty the FOR loop will be performed zero times.

Syntax:

```
50 for_expr = foreach_expr | forloop_expr .
51 foreach_expr = FOR EACH variable_id IN foreach_in_clause_arg { AND
   variable_id IN foreach_in_clause_arg } [ from_clause ] [ where_clause ]
   RETURN map_attr_assgnmt_expr ';' .
52 foreach_in_clause_arg = attribute_reference | view_attribute_reference
   | extent_reference .
54 forloop_expr = FOR repeat_control RETURN map_attr_assgnmt_expr ';' .
```

Rules and restrictions:

a) The target attribute of the attribute assignment statement where the FOR expression is used in shall be of type AGGREGATE.

10.5 Conditional expression

This concept is introduced for MAP declarations so that a specified expression is assigned to a target attribute under some condition (or, else another expression is assigned). The conditional expressions can be nested.

Syntax:

```
73 map_cond_attr_expr = IF boolean_expression THEN map_attr_assgnmt_expr
   [ ELSE map_attr_assgnmt_expr ] END_IF ';' .
```

10.6 CASE expression

The CASE expression is similar to the CASE statement of EXPRESS.

EXAMPLE 35 — CASE expression.

```
MAP my_approval
FROM a : approval
SELECT
  status := CASE a.status OF
              'approved'     : 1;
              'not approved' : -1;
              'indetermined' : 0;
              OTHERWISE      : 2;
            END_CASE;
END_MAP;
```

Syntax:

```
40 case_expr = CASE selector OF { case_expr_action } [ OTHERWISE ':'
   expression ] END_CASE ';' .
41 case_expr_action = case_label { ',' case_label } ':' expression .
71 map_case_expr = CASE selector OF { map_case_expr_action } [ OTHERWISE
   ':' map_attr_assgnmt_expr ] END_CASE ';' .
72 map_case_expr_action = case_label { ',' case_label } ':'
   map_attr_assgnmt_expr .
```

11. Built-in functions

11.1 Extent - general function

```
FUNCTION EXTENT ( R : STRING ) : SET OF GENERIC;
```

The EXTENT function returns the population of instances of the type specified by the parameter.

Parameters:

a) R is a string that contains the name of a entity data type or view data type. Such names are qualified by the name of the schema which contains the definition of the type ('SCHEMA.TYPE').

Result: A set containing all instances of the entity data type or view data type specified in the parameter. It is an error to specify as the parameter a type which is neither a view data type nor an entity data type defined in a source schema.

12. Scope and visibility

An EXPRESS-X declaration creates an identifier that can be used to reference the declared item in other parts of the schema_view (or in other schema_views). Some EXPRESS-X constructs implicitly declare items, attaching identifiers to them. An item is said to be visible in those areas where an identifier for a declared item may be referenced. An item may only be referenced where its identifier is visible. For the rules of visibility, see clause10.2 For further information on referring to items using their identifiers, see clause12.

Certain EXPRESS-X items define a region (block) of text called the scope of the item. This scope limits the visibility of identifiers declared within it. Scope can be nested; that is, an EXPRESS-X item which establishes a scope may be included within the scope of another item. There are constraints on which items may appear within a particular EXPRESS-X item's scope.

For each of the items specified in table2 below the following subclauses specify the limits of the scope defined, if any, and the visibility of the declared identifier both in general terms and with specific details.

Table 2 — Scope and identifier defining items

Item	Scope	Identifier
view attribute		•
view	•	•
partition	•	•
schema_view	•	•

12.1 Scope rules

The general scope rules are as defined in ISO10303-11:1994.

12.2 Visibility rules

The general visibility rules are as defined in ISO10303-11:1994.

12.3 Explicit item rules

The following language elements provide more detail on how the general scoping and visibility rules apply to the various EXPRESS-X items.

12.3.1 Schema_view

Visibility: A schema_view identifier is visible to all other schema_views.

Scope: A schema_view declaration defines a new scope. This scope extends from the keyword SCHEMA_VIEW to the keyword END_SCHEMA_VIEW that terminates that schema_view declaration.

Declarations: The following EXPRESS-X items may declare identifiers within the scope of a schema_view declaration:

- constant;
- function;
- map;
- procedure;
- rule;
- type_map;
- view.

12.3.2 View

Visibility: A view identifier is visible in the scope of the function, procedure, rule, or schema_view in which it is declared. A view identifier remains visible within inner scopes which redeclare that identifier.

Scope: A view declaration defines a new scope. This scope extends from the keyword VIEW to the keyword END_VIEW which terminates that entity declaration.

Declarations: The following EXPRESS-X items may declare identifiers within the scope of a view declaration:

- view attribute;
- partition label.

12.3.3 View partition label

Visibility: A partition label is visible in the scope of the view in which it is declared.

12.3.4 View attribute identifier

Visibility: A view attribute identifier is visible in the scope of the view in which it is declared.

13. Interface specification

This clause specifies the constructs that enable items declared in one schema, schema_view, or schema_map to be visible in another schema_view or schema_map. The REFERENCE specification enables item visibility.

Syntax:

```
69 map_interface_spec = IMPORT_MAPPING schema_map_or_view_ref_or_rename [
   REFERENCE resource_or_rename { ',' resource_or_rename } ] ';' .
89 schema_map_or_view_ref_or_rename = schema_map_ref_or_rename |
   schema_view_ref_or_rename .
90 schema_map_ref_or_rename = [ schema_map_alias_id ':' ] schema_map_ref
   .
97 schema_view_ref_or_rename = [ schema_view_alias_id ':' ]
   schema_view_ref .
```

A foreign declaration is any declaration which appears in a foreign schema, schema_view, or schema_map (which is not the current schema_view or schema_map).

A foreign EXPRESS or EXPRESS-X item may be given a new name in the current schema_view or schema_map. The item shall be referred to in the current schema by the new name if given following the AS keyword. This can be used in order to rename EXPRESS items that would otherwise use EXPRESS-X reserved words as their identifier.

13.1 Reference interface specification

A REFERENCE specification enables the following items, declared in a foreign schema, schema_view, or schema_map, to be visible in the current schema_view or schema_map:

– View;

– Map;

– Type_map;

– Constant;

– Entity;

– Function;

– Procedure;

– Type.

The REFERENCE specification gives the name of the foreign schema, schema_view, or schema_map, and optionally the names of EXPRESS or EXPRESS-X items declared therein. If there are no names specified, all the items declared in the foreign schema, schema_view, or schema_map are visible within the current schema_view or schema_map.

The schema_ref may be an EXPRESS-X reserved word that is not also an EXPRESS reserved word.

Syntax:

```
 82  reference_clause_extended = REFERENCE FROM foreign_ref [ '('
     resource_or_rename { ',' resource_or_rename } ')' ] ';' .
 53  foreign_ref = schema_ref | schema_view_ref | schema_map_ref .
```

Rules and restrictions:

13.2 Implicit interfaces

13.3 SCHEMA_MAP interfaces

A schema_map interface specification identifies the source and target schema and allows items defined in these schema to be visible within the schema map.

Syntax:

```
 87  schema_map_decl = SCHEMA_MAP schema_map_id target_interface_spec {
     target_interface_spec } source_interface_spec { source_interface_spec
     } { map_interface_spec } { type_mapping_stmt } [ constant_decl ]
     schema_map_body_element_list END_SCHEMA_MAP ';' .
```

13.3.1 Source schema interface

The source schema interface specifies the name of the source schema.

Syntax:

```
100  source_interface_spec = SOURCE schema_ref_or_rename [ REFERENCE
     resource_or_rename { ',' resource_or_rename } ] ';' .
```

13.3.2 Target schema interface

The target schema interface specifies the name of the target schema.

Syntax:

```
108  target_interface_spec = TARGET schema_ref_or_rename [ REFERENCE
     resource_or_rename { ',' resource_or_rename } ] ';' .
```

13.3.3 Map interface

The map interface specifies how one SCHEMA_MAP may reference another.

Syntax:

```
 69  map_interface_spec = IMPORT_MAPPING schema_map_or_view_ref_or_rename [
     REFERENCE resource_or_rename { ',' resource_or_rename } ] ';' .
```

Annex A
(normative)
EXPRESS-X language syntax

This annex defines the lexical elements of the language and the grammar rules that these elements shall obey.

NOTE — This syntax definition will result in ambiguous parsers if used directly. It has been written so as to convey information regarding the use of identifiers. The interpreted identifiers define tokens that are references to declared identifiers, and therefore should not resolve to simple_id. This requires a parser developer to enable identifier reference resolution and return the required reference token to a grammar rule checker.

All of the grammar rules of EXPRESS specified in annex A of ISO 10303-11:1994 are also grammar rules of EXPRESS-X. In addition, the grammar rules specified in the remainder of this annex are grammar rules of EXPRESS-X.

A.1 Tokens

The following rules specify the tokens used in EXPRESS-X. Except where explicitly stated in the syntax rules, no white space or remarks shall appear within the text matched by a single syntax rule in the following clauses.

A.1.1 Keywords

This subclause gives the rules used to represent the keywords of EXPRESS-X.

NOTE — This subclause follows the typographical convention that each keyword is represented by a syntax rule whose left hand side is that keyword in uppercase.

NOTE — All the keywords of EXPRESS are also keywords of EXPRESS-X

```
 1  BETWEEN = 'between' .
 2  CHOICE = 'choice' .
 3  CREATE = 'create'.
 4  EACH = 'each'.
 5  ELSIF = 'elsif' .
 6  END_CHOICE = 'end_choice' .
 7  END_CREATE = 'end_create'.
 8  END_FOR = 'end_for'.
 9  END_MAP = 'end_map'.
10  END_SCHEMA_MAP = 'end_schema_map'.
11  END_SCHEMA_VIEW = 'end_schema_view'.
12  END_TYPE_MAP = 'end_type_map'.
13  END_VIEW = 'end_view'.
```

```
14  EXTENT = 'extent' .
15  IDENTIFIED_BY = 'identified_by'.
16  IMPORT_MAPPING = 'import_mapping'.
17  MAP = 'map'.
18  PARTITION = 'partition'.
19  SCHEMA_MAP = 'schema_map'.
20  SCHEMA_VIEW = ’schema_view’.
21  SOURCE = 'source'.
22  TARGET = 'target'.
23  TYPE_MAP = 'type_map'.
24  VIEW = 'view'.
```

A.1.2 Character classes

```
25  digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
26  letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
           | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
           | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .
27  simple_id = letter { letter  | digit  | '_' } .
```

A.1.3 Interpreted identifiers

NOTE — All interpreted identifiers of EXPRESS are also interpreted in EXPRESS-X

```
28  instance_ref = instance_id .
29  network_ref = network_id .
30  partition_ref = partition_id .
31  schema_map_ref = schema_map_id .
32  schema_view_ref = schema_view_id .
33  source_schema_ref = schema_ref .
34  target_schema_ref = schema_ref .
35  view_attribute_ref = view_attribute_id .
36  view_ref = view_id .
```

A.2 Grammar rules

```
37  attr_assgnmt_expr = type_assgnmt_expr | view_attr_assgnmt_expr
                      | map_attr_assgnmt_expr .
38  attribute_reference =   attribute_ref
    | primary_extended attribute_qualifier  .
39  binding_decl = [ from_clause ] [ where_clause ]
                   [ identified_by_clause ].
40  boolean_expression = expression .
```

```
41 choice_case_expr = CHOICE selector OF
                        case_label {',' case_label }
                          THEN attr_assgnmt_expr ';'
                        { case_label {',' case_label }
                          THEN attr_assgnmt_expr ';' }
                        [ ELSE attr_assgnmt_expr ';']
                      END_CHOICE .
42 choice_expr = choice_if_expr | choice_case_expr .
43 choice_if_expr = CHOICE logical_expression  THEN attr_assgnmt_expr ';'
                   { ELSIF logical_expression THEN attr_assgnmt_expr ';' }
                    [ ELSE attr_assgnmt_expr ';' ]
                    END_CHOICE .
44 complex_entity_spec = entity_reference '&' entity_reference
                                          { '&' entity_reference } .
45 create_map_decl = CREATE instance_id ':' target_entity_reference ';'
                     [ WHERE logical_expression ';' ]
                     map_attr_decl_stmt_list
                     END_CREATE ';' .
46 entity_instantiation_loop = FOR instantiation_loop_control ';' .
47 entity_qualifier = '.' entity_ref .
48 entity_reference = [ ( schema_map_ref | schema_view_ref
                          | schema_ref ) '.' ] entity_ref .
49 extent_reference = source_entity_reference | view_reference .
50 for_expr = foreach_expr | forloop_expr .
51 foreach_expr = FOR EACH variable_id IN foreach_in_clause_arg
                    { AND variable_id IN foreach_in_clause_arg }
                  [ from_clause ] [ where_clause ]
                  RETURN map_attr_assgnmt_expr ';' .
52 foreach_in_clause_arg = attribute_reference
                           | view_attribute_reference | extent_reference .
53 foreign_ref = schema_ref | schema_view_ref | schema_map_ref .
54 forloop_expr = FOR repeat_control RETURN map_attr_assgnmt_expr ';' .
55 from_clause = FROM source_parameter { ';' source_parameter } ';'.
56 source_parameter = source_parameter_id { ',' source_parameter_id } ':'
   extent_reference.
57 identified_by_clause = IDENTIFIED_BY expression { ',' expression } ';'.
58 inline_view_decl = VIEW from_clause [ where_clause ]
                     [ view_project_clause ] END_VIEW ';' .
59 instance_id = simple_id .
60 instance_qualifier = '.' instance_ref .
61 instantiation_foreach_control = EACH variable_id
                                  IN source_attribute_reference
                                  [ INDEXING variable_id ]
                                  { AND variable_id
                                  IN source_attribute_reference
                                  [ INDEXING variable_id ] }  .
62 instantiation_loop_control = instantiation_foreach_control
                              | repeat_control .
```

```
63  map_attr_assgnmt_expr = expression | choice_expr | for_expr
                          | map_call .
64  map_attr_decl_stmt_list = map_attribute_declaration
                            { map_attribute_declaration } .
65  map_attribute_declaration = [ target_parameter_ref
                                  [ index_qualifer ]
                                  [ group_qualifier ] '.' ]
                                attribute_ref [ index_qualifier ] ':='
                                map_attr_assgnmt_expr ';' .
66  map_call = target_parameter_ref [ map_or_partition_qualification ]
               '(' expression { ',' expression } ')' .
67  map_decl = MAP map_id AS target_parameter { target_parameter }
               ( (map_decl_body { map_partitions }) | map_decl_body )
               END_MAP ';' .
68  map_decl_body = [subtype_of_clause] binding_decl
                  { entity_instantiation_loop }
                   map_project_clause .
69  map_interface_spec = IMPORT_MAPPING schema_map_or_view_ref_or_rename
                       [ REFERENCE resource_or_rename
                         { ',' resource_or_rename } ] ';' .
70  map_or_partition_qualification = '@' map_ref |
    | '@'  map_ref '.'  partition_ref .
71  map_partition =  PARTITION partition_id ':' map_decl_body .
72  map_partitions = map_partition { map_partition } .
73  map_project_clause = (SELECT map_attr_decl_stmt_list) | ( RETURN
    expression ) .
74  map_reference = [ schema_map_ref '.' ] map_ref .
75  map_id = simple_id .
76  map_ref = map_id .
77  partition_id = simple_id .
78  partition_qualification = '\' partition_ref .
79  primary_extended = qualifiable_factor_extended { qualifier_extended } .
80  qualifiable_factor_extended =   qualifiable_factor  | schema_map_ref  |
    schema_view_ref  | view_ref  | map_call  | view_call  |
    view_attribute_ref  | instance_ref  .
81  qualifier_extended = qualifier | instance_qualifier | entity_qualifier
                       | view_attribute_qualifier .
82  reference_clause_extended = REFERENCE FROM foreign_ref
                      [ '(' resource_or_rename
                           { ',' resource_or_rename } ')' ] ';' .
83  schema_alias_id = schema_id .
84  schema_map_alias_id = schema_map_id .
85  schema_map_body_element = function_decl | procedure_decl
                            | view_decl | map_decl | create_map_decl .
86  schema_map_body_element_list = schema_map_body_element
                                 { schema_map_body_element } .
```

```
 87  schema_map_decl = SCHEMA_MAP schema_map_id
                       target_interface_spec { target_interface_spec }
                       source_interface_spec { source_interface_spec }
                       { map_interface_spec }
                       { type_mapping_stmt }
                       [ constant_decl ]
                       schema_map_body_element_list
                       END_SCHEMA_MAP ';' .
 88  schema_map_id =  simple_id .
 89  schema_map_or_view_ref_or_rename = schema_map_ref_or_rename
                                      | schema_view_ref_or_rename .
 90  schema_map_ref_or_rename =  [ schema_map_alias_id ':' ]
                                 schema_map_ref .
 91  schema_ref_or_rename = [ schema_alias_id ':' ] schema_ref .
 92  schema_view_alias_id = schema_view_id .
 93  schema_view_body_element = function_decl | procedure_decl | view_decl .
 94  schema_view_body_element_list =   schema_view_body_element   {
     schema_view_body_element }  .
 95  schema_view_decl = SCHEMA_VIEW schema_view_id {
     reference_clause_extended } [ constant_decl ]
     schema_view_body_element_list END_SCHEMA_VIEW ';' .
 96  schema_view_id = simple_id .
 97  schema_view_ref_or_rename = [ schema_view_alias_id ':' ]
                                 schema_view_ref .
 98  source_attribute_reference = attribute_reference  |
     view_attribute_reference .
 99  source_entity_reference = entity_reference .
100  source_interface_spec = SOURCE schema_ref_or_rename
                             [ REFERENCE resource_or_rename
                               { ',' resource_or_rename } ]  ';' .
101  source_parameter_id = parameter_id .
102  subtype_of_clause =  SUBTYPE OF '(' view_or_map_reference
                          { ',' view_or_map_reference } ')' .
103  syntax = schema_map_decl | schema_view_decl .
104  target_parameter =  [ target_parameter_id
                           { ',' target_parameter_id } ':' ]
                         [ AGGREGATE [ bound_spec ] OF ]
                         target_entity_reference ';' .
105  target_parameter_id = parameter_id .
106  target_parameter_ref = target_parameter_id .
107  target_entity_reference =   entity_reference  | complex_entity_spec  |
     target_schema_ref '.' '(' complex_entity_spec ')'  .
108  target_interface_spec = TARGET schema_ref_or_rename
                             [ REFERENCE resource_or_rename
                               { ',' resource_or_rename } ]  ';' .
109  type_assgnmt_expr = expression | choice_expr .
110  type_map_stmt_body = [ schema_ref '.' ]
                          base_type ':=' type_assgnmt_expr ';' .
```

```
111 type_mapping_stmt = TYPE_MAP
                          type_reference FROM type_reference ';'
                          type_map_stmt_body type_map_stmt_body
                        END_TYPE_MAP ';' .
112 type_reference = [ schema_ref '.' ] type_ref .
113 view_attr_assgnmt_expr = expression | choice_expr | inline_view_decl
                            | view_call .
114 view_attr_decl_stmt_list = view_attribute_decl
                              { view_attribute_decl } .
115 view_attribute_decl = view_attribute_id ':' [ source_schema_ref '.' ]
                          base_type ':=' view_attr_assgnmt_expr ';' .
116 view_attribute_id = simple_id .
117 view_attribute_qualifier = '.' view_attribute_ref .
118 view_attribute_reference = view_attribute_ref
    | primary_extended view_attribute_qualifier  .
119 view_call = view_reference [ partition_qualification ]
              '(' expression { ',' expression } ')' .
120 view_decl = VIEW view_id [ : base_type ][ supertype_rule ] [
    subtype_of_clause ] ';'
              ( view_partitions  | view_decl_body )
              END_VIEW ';' .
121 view_decl_body =  binding_decl view_project_clause .
122 view_id = simple_id .
123 view_or_map_reference =  view_reference | map_reference .
124 view_partition =   PARTITION partition_id ';' view_decl_body .
125 view_partitions = view_partition { view_partition } .
126 view_project_clause =  ( SELECT view_attr_decl_stmt_list) | ( RETURN
    expression ) .
127 view_reference = [ ( schema_map_ref | schema_view_ref ) '.' ]
                    view_ref .
```

A.3 EXPRESS Syntax

```
128 add_like_op = '+'| '-' | OR | XOR .
129 bound_1 = numeric_expression .
130 bound_2 = numeric_expression .
131 bound_spec = '[' bound_1 ':' bound_2 ']' .
132 built_in_constant = CONST_E | PI | SELF | '?' .
133 built_in_function = ABS | ACOS | ASIN | ATAN | BLENGTH | COS | EXISTS
                      | EXP | FORMAT | HIBOUND | HIINDEX | LENGTH | LOBOUND
                      | LOINDEX | LOG | LOG2 | LOG10 | NVL | ODD | ROLESOF
                      | SIN | SIZEOF | SQRT | TAN | TYPEOF | USEDIN | VALUE
                      | VALUE_IN | VALUE_UNIQUE .
134 constant_factor = built_in_constant | constant_ref .
135 enumeration_reference = [ type_ref '.' ] enumeration_ref .
136 expression = simple_expression [ rel_op_extended simple_expression ] .
```

```
137  factor = simple_factor [ '**' simple_factor ] .
138  logical_expression =  expression .
139  numeric_expression = simple_expression .
140  repeat_control = [ increment_control ] [ while_control ]
                      [ until_control ] .
141  simple_factor = aggregate_initializer | entity_constructor
                   | enumeration_reference | interval | query_expression
                   | ( [ unary_op ] ( '(' expression ')'  | primary ) ) .
```

A.4 Cross reference listing

(informative)
Bibliography

EXPRESS-V language (ISO TC184/SC4/WG5 N251).

EXPRESS-M language (ISO TC184/SC4/WG5 N243).

BRITTY language.

Wirth, Niklaus, *"What can we do about the unnecessary diversity of notations for syntactic definitions?,"* Communications of the ACM, November 1977, v. 20, no. 11, p. 822.

Annex B
(normative)
EXPRESS-X to EXPRESS Tranformation Algorithm

This annex describes how a collection of view declarations may be transformed into a collection of EXPRESS entity declarations suitable for representing the results of an EXPRESS-X execution. The transformation is described as an algorithm taking the text of a view declaration as input and producing the text of an entity declaration as output. The algorithm is given here for specification purposes only, not to prescribe a particular implementation.

The transformed entities are assumed to exist in a uniquely named schema, into which all necessary foreign declarations have been interfaced.

Algorithm:

a) If the view declaration is a SELECT view (i.e., does not define any view attributes), skip the declaration.

b) Change the keyword VIEW to ENTITY.

c) Delete entirely any FROM ,WHERE, and/or IDENTIFIED_BY clauses. Delete only WHERE clauses in the header; do not delete constraint where clauses.

d) Delete the keyword SELECT.

e) If the view declaration contains partitions, delete entirely all but the first partition declaration, and delete the keyword PARTITION and the partition identifier (if any) from the first partition declaration.

f) Delete the assignment operator and expression for each view attribute.

g) Change the keyword END_VIEW to END_ENTITY.

EXAMPLE 36 —

```
VIEW a ABSTRACT SUPERTYPE;
PARTITION one:
FROM b:one, c:two
WHERE cond1;
      cond2;
SELECT
  x : attr1 := expression1;
  y : attr2 := expression2;
PARTITION two:
FROM d:two, e:three
WHERE cond3;
      cond4;
SELECT
  x : attr1 := expression3;
  y : attr2 := expression4;
END_VIEW;
```

is transformed into the following EXPRESS entity declaration:

```
ENTITY a ABSTRACT SUPERTYPE;
  x : attr1;
  y : attr2;
END_ENTITY;
```

EXAMPLE 37 —

```
VIEW b SUBTYPE OF (a);
PARTITION one:
WHERE cond5;
SELECT
  z : attr3 := expression5;
PARTITION two:
WHERE cond6;
SELECT
  z : attr3 := expression6;
WHERE
  WR2 : rule_expression2;
END_VIEW;
```

is transformed into the following EXPRESS entity declaration:

```
ENTITY b SUBTYPE OF (a);
  z : attr3;
WHERE
  WR2 : rules_expression2;
END_ENTITY;
```

Annex C

13.3.3.1 Push mapping

An implementation shall be said to be a push mapping implementation if it meets all of the following criteria:

– The mapping engine accepts one or more source data sets, and produces one or more output data sets.

– The output data sets are derived from the input data sets by the execution and evaluation of all of the VIEW and MAP declarations.

– Every instance in the source data sets is mapped as specified in the mapping schema into the output data sets.

13.3.3.2 Pull mapping

An implementation shall be said to be a pull mapping implementation if it meets all of the following criteria:

– The mapping engine accepts one or more source data sets.

– Specified target data instances, and only those specified, are derived on demand from the input data sets by the execution and evaluation of the appropriate VIEW or MAP declarations.

NOTE — This part of ISO 10303 does not define how VIEW / MAP declarations are selected for pull mapping.

13.3.3.3 Support of constraint checking

An implementation shall be said to support constraint checking if it implements the concepts described in clause 9.6 of ISO 10303-11:1994 against entity instances in target populations and against view instances in the view extents.

NOTE — The evaluation of constraints has no effect on execution.

Propagation of updates is not possible in situations where any of the following hold:

– The view / target entity is derived from / mapped to two or more source entities by applying a join operation. (For example, the view / target entity `person_in_dept` corresponds to the source entities `person` and `department` where the join condition `person.id = department.person_id` evaluates to true.)

– Duplicates (with respect to value equivalence of attributes) which exist in the source data are eliminated in the view / target data.

– View / target attributes are derived from / mapped to source schema elements by applying mathe-

matical expressions that are not mathematically invertible.

– The view / target schema defines additional subtypes which do not exist in the source schema(s).

– Subtypes which are defined in the source schema(s) are projected (i.e., not contained) in the view / target schema.

– The sort order of source attributes of type AGGREGATE is eliminated in the view / target schema.

– Duplicates (with respect to value equivalence) of elements of source attributes of type AGGREGATE are eliminated in the view / target schema.

– A single source entity corresponds to a network of interconnected view / target entities (by relationships or equivalence of attribute values[1]).

1. The latter kind of relationship is comparable to primary key - foreign key relationships in the relational data model.